



International Conference on Computational Science, ICCS 2013

Eden's Bees: Parallelizing Artificial Bee Colony in a Functional Environment[☆]

Fernando Rubio*, Alberto de la Encina, Pablo Rabanal, Ismael Rodríguez

*Dpto. Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid
Madrid 28040, Spain*

Abstract

Due to the proliferation of multicore computers, most users own hardware that allows to speedup the execution of programs by running them in parallel. However, in practice it is not trivial to take advantage of such parallel architectures, because the programmer needs to take care of too many low level details. This is also true in problems with a high degree of inherent parallelism, like many bioinspired metaheuristics. In this paper we simplify the parallelization of one of such metaheuristics, namely Artificial Bee Colony, by using a functional language, Eden, to implement a parallel skeleton to deal with it. Once the skeleton is defined, the user only needs to provide a concrete fitness function, while all the low level parallel details are done automatically by the skeleton.

Keywords: Artificial Bee Colony; Swarm Intelligence; Parallel Programming; Functional Programming.

1. Introduction

Many bioinspired methods are based on using several simple entities which search for a reasonable solution (somehow) independently. This is the case of Artificial Bee Colony [1, 2, 3] (ABC), where many simple bees search for the optimal solution by using both their local information and the information about the position of other bees of the colony. Thus, bees are partially independent, and we can take advantage of this fact to parallelize ABC programs. Unfortunately, providing good parallel implementations for each specific ABC program can be tricky and time-consuming for the programmer.

The higher-order nature of functional languages, where programs (functions) can be treated as any other kind of program data, make them very useful to implement generic programming solutions which can be trivially reused for other problems. This is even clearer in parallel programming. Since programs can trivially refer to themselves, the coordination of parallel subcomputations can be defined by using the same constructions as in the rest of the program. This enables the construction of *skeletons* [4], which are generic implementations of parallel schemes which can be invoked in run-time to immediately obtain a parallelization of a given sequential program.

[☆]Research partially supported by project TIN2012-39391-C04-04.

*Corresponding author. Tel.: +34-91-394-7629 ; fax: +34-91-394-7636.

E-mail address: fernando@sip.ucm.es.

The skeleton defines how subcomputations coordinate in parallel, and the skeleton user just has to define (part of) the subcomputations.

During the last years, several parallel functional languages have been proposed (see e.g. [5, 6, 7, 8, 9, 10]). In this paper we show how to use one of them to simplify the development of parallel versions of Swarm Intelligence methods. In particular, we use the language Eden [11, 8] to create a generic skeleton implementing the parallelization of ABC. Then, we use the skeleton and report some experimental results. We observe that, despite of the low effort required by programmers to use these skeletons, empirical results show that skeletons reach reasonable speedups.

The rest of the paper is structured as follows. Next, we briefly describe the parallel language Eden. In Section 3 we introduce the ABC method. Then, in Section 4, we present how to develop a higher-order sequential Haskell function dealing with ABC. Afterwards, in Section 5 we introduce a skeleton that parallelizes such higher-order function. Finally, Section 6 contains our conclusions and lines for future work.

2. Introduction to Eden

Eden [11, 8] is a parallel extension of Haskell [12], the *de facto* standard of the lazy-evaluation¹ functional programming community. Haskell is a strongly typed language including polymorphism, higher-order programming features and lazy order of evaluation of expressions.

Eden introduces parallelism by adding syntactic constructs to define and instantiate processes explicitly. It is possible to define a new *process abstraction* p by applying the predefined function *process* to any function $\lambda x \rightarrow e$, where variable x will be the input of the process, while the behavior of the process will be given by expression e . Process abstractions are similar to functions – the main difference is that the former, when instantiated, are executed in parallel. From the semantics point of view, there is no difference between process abstractions and function definitions. The differences between processes and functions appear when they are invoked. Processes are invoked by using the predefined operator $\#$. For instance, in case we want to create a *process instantiation* of a given process p with a given input data x , we write $(p \# x)$. Note that, from a syntactical point of view, this is similar to the *application* of a function f to an input parameter x , which is written as $(f \ x)$.

Therefore, when we refer to a *process* we are not referring to a syntactical element but to a new *computational environment*, where the computations are carried out in an autonomous way. Thus, when a *process instantiation* $(e_1 \# e_2)$ is invoked, a new *computational environment* is created. The new process (the child or instantiated process) is fed by its creator by sending the value for e_2 via an input channel, and returns the value for $e_1 e_2$ (to its parent) through an output channel.

Let us remark that, in order to increase parallelism, Eden employs pushing instead of pulling of information. That is, values are sent to the receiver before it actually demands them. In addition to that, once a process is running, only fully evaluated data objects are communicated. The only exceptions are *streams*, which are transmitted element by element. Each stream element is first evaluated to full normal form and then transmitted. Concurrent threads trying to access not yet available input are temporarily suspended. This is the only way in which Eden processes synchronize. Notice that process creation is explicit, but process communication (and synchronization) is completely implicit.

2.1. Eden Skeletons

Process abstractions in Eden are not just annotations, but first class values which can be manipulated by the programmer (passed as parameters, stored in data structures, and so on). This facilitates the definition of skeletons as higher order functions. Next we illustrate, by using a simple example, how skeletons can be written in Eden. More complex skeletons can be found in [11]. The most simple skeleton is *map*. Given a list of inputs xs and a function f to be applied to each of them, the sequential specification in Haskell is as follows:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

¹Lazy evaluation is an evaluation strategy which delays the evaluation of each expression until the exact moment when it is actually required.

The first line of the definition is optional, and it represents the type declaration of the function: it indicates that its first parameter has type $a \rightarrow b$, denoting that the first parameter is a function that receives values of type a and returns values of type b . The second parameter is a list of elements of type a , and the result is again a list, but in this case of elements of type b . Regarding the body of the function (second line), it can be read as *for each element x belonging to the list xs , apply function f to that element*. This can be trivially parallelized in Eden. In order to use a different process for each task, we will use the following approach:

```
map_par f xs = [pf # x | x <- xs] 'using' spine    where pf = process f
```

The process abstraction pf wraps the function application ($f\ x$). It determines that the input parameter x as well as the result value will be transmitted through channels. The *spine* strategy (see ([13] for details) is used to eagerly evaluate the spine of the process instantiation list. In this way, all processes are immediately created. Otherwise, they would only be created on demand.

Let us remark that Eden's compiler has been developed by extending the GHC Haskell compiler. Hence, it reuses GHC's capabilities to interact with other programming languages. Thus, Eden can be used as a coordination language, while the sequential computation language can be, for instance, C.

3. Artificial Bee Colony Algorithm

An interesting branch of bioinspired algorithms is Swarm Intelligence [14, 15, 16, 17], where the behavior of simple agents that interact with the environment, as well as among them, leads to the appearance of an *intelligent* global behavior. In 2005, Karaboga developed an algorithm for solving numeric optimization problems which simulates the behavior of bees when they look for the best food source, called Artificial Bee Colony (ABC) algorithm [2, 3].

ABC uses two kind of bees to explore the solution space: employed bees and onlooker bees. *Employed bees* are randomly distributed among the solution space, and explore those areas in their neighborhood searching for the best nectar source, that is the best solution, depending on their own experience and the experience of their beehive mates. *Onlooker bees* produce new solutions without using any experience, and try to find promising food areas in order to attract employed bees to explore around the new food source found. In this way, by using employed and onlooker bees, ABC combines local search with global search methods, trying to balance the exploration and exploitation process.

Let us suppose we have to solve an optimization problem where the function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ has to be minimized in the range $[bot, up]$, where *bot* and *up* are the lower and upper bounds of the search space. Each bee i of the colony has the following attributes: a position value x_i , the function value $f(x_i)$, the fitness value $fit(x_i)$, the *normalized* fitness p_i (later we will see how it is defined and used), and the *scouting counter* L_i , which controls when an exploration area has to be abandoned. Furthermore, values associated to the best nectar source found are stored: x_{best} , $f(x_{best})$, and $fit(x_{best})$.

When an employed bee is moved, a new position value v_i is created depending on x_i , x_j , and some randomness, where x_j is the position of another employed bee in the swarm. If the new solution v_i improves the old one x_i , the solution x_i is replaced with v_i . However, an onlooker bee produces a new random solution v_i without relying on anything, and compares the solution found with one of the solutions found by an employed bee j . Again, if the solution found v_i improves the solution x_j , where index j is randomly selected depending on the fitness of all the solutions found, then the solution x_j is replaced with v_i .

The main steps of the ABC algorithm are shown below:

```
initializeFoodSources()
Repeat
  moveEmployedBees()
  normalizeFitness()
  moveOnlookerBees()
  replaceAbandonedSolutions()
Until endingCondition()
```

First of all we have to set the colony size CS , where $\frac{CS}{2}$ will be employed bees and $\frac{CS}{2}$ will be onlooker bees; and the scouting limit L , which is defined as follows: $L = \frac{CS \cdot D}{2}$, where $D = n$ is the dimension of the function f to be optimized.

Next we explain each step of the algorithm in detail. In the `initializeFoodSources()` phase, the position x_i of each employed bee is randomly created in the range $[bot, up]$, and the values of $f(x_i)$ and $fit(x_i)$ are calculated. The fitness function fit is defined as: $fit(x_i) = \frac{1}{1+f(x_i)}$ if $f(x_i) \geq 0$, and $fit(x_i) = 1 + abs(f(x_i))$ otherwise. After initializing the first bee, the best solution found is also initialized: $x_{best} = x_1$, $f(x_{best}) = f(x_1)$, and $fit(x_{best}) = fit(x_1)$. For the rest of employed bees ($i > 1$), if $fit(x_i) > fit(x_{best})$ then $x_{best} = x_i$, $f(x_{best}) = f(x_i)$, and $fit(x_{best}) = fit(x_i)$.

The body of the Repeat loop is executed until the `endingCondition()` is satisfied. Usually, the `endingCondition()` limits the number of iterations performed, the time consumed, or the fitness adequation. The first step of the loop, `moveEmployedBees()`, consists in moving all the employed bees in the following manner. For each bee, a new solution v_i is produced by changing one randomly chosen dimension of x_i . For this purpose, the formula $v_i = x_i + \Phi \cdot (x_i - x_j)$ is used, where Φ is a random number in the range $[-1, 1]$ and $j \in [1, \frac{CS}{2}]$ is a random selected index. After that, $f(v_i)$ and $fit(v_i)$ are computed. If $fit(v_i) > fit(x_i)$, the solution is replaced: $x_i = v_i$, $f(x_i) = f(v_i)$, $fit(x_i) = fit(v_i)$, and L_i is set to L . In other case, L_i is decreased by one unit. The best solution is also updated if $fit(x_i) > fit(x_{best})$, as we explained in the previous paragraph.

In the `normalizeFitness()` step, the fitness of each bee is normalized into $[0, 1]$ as follows:

$$p_i = \frac{fit(x_i)}{\sum_{k=1}^{CS/2} fit(x_k)}$$

Next, onlooker bees are moved in the `moveOnlookerBees()` phase. Each onlooker bee selects the solution x_i (of an employed bee) depending on p_i and creates a new solution v_i by changing one dimension of x_i . Again, if $fit(v_i) > fit(x_i)$, the solution of the corresponding employed bee is replaced: $x_i = v_i$, $f(x_i) = f(v_i)$, $fit(x_i) = fit(v_i)$, and $L_i = L$. Else, $L_i = L_i - 1$. The best solution is also updated if $fit(x_i) > fit(x_{best})$. Note that the new value v_i does not depend on any other value, and it is created in a completely random way.

Finally, if there exists an abandoned solution x_i , that is, a solution with $L_i < 0$, a new solution is randomly generated to replace it in the `replaceAbandonedSolutions()` step, and its corresponding $f(x_i)$ and $fit(x_i)$ are calculated.

4. Generic ABC in Haskell

In order to provide a general scheme to deal with ABC algorithms, we can take advantage of the higher-order nature of Haskell. Due to the fact that Haskell functions are first-class citizens, we can use them as parameters of other functions. Thus, we can define a higher-order function `beesSEQ` having as input parameter the concrete fitness function used in the problem. In addition to the fitness function, we also need to consider other parameters like the number of bees to be used, the number of iterations to be executed by the program, the boundings of the search space, and the scouting parameter described in Section 3. Moreover, in order to implement it in a pure functional language like Haskell, we need an additional parameter to introduce randomness. Note that Haskell functions cannot produce side-effects, so they need an additional input parameter to be able to obtain different results in different executions. Taking into account these considerations, the type of the higher-order Haskell function dealing with ABC algorithms is the following:

```
beesSEQ :: RandomGen a => a           -- Random generator
    -> Scouting                       -- Scouting counter parameter
    -> Int                            -- Number of bees to be used
    -> Int                            -- Maximum number of iterations
    -> (Position -> Double)           -- Fitness function
    -> Boundings                     -- Search space boundaries
    -> (Double, Position)             -- Value and position of best fitness
```

Let us remark that we use as parameter the fitness function, but not the function value. The reason is that each function can be obtained from the other one. Thus, we only need one of them. Note also that the type `Position` should be able to deal with arbitrary dimensions. Thus, the simplest and more general solution to define a position is to use a list of real numbers, where the length of the list is equal to the number of dimensions of the search space. Analogously, the type `Boundings` should contain a list of lower and upper bounds for each of the dimensions. Thus, it can be described by a list of pairs of real numbers. Finally, the type `Scouting` is even simpler, as it only contains an integer describing the concrete scouting counter used in the algorithm:

```
type Position = [Double]
type Boundings = [(Double,Double)]
type Scouting = Int
```

Once the input parameters are defined, it is time to define the body of the function. First, we use function `initializeBees` to randomly create the list of initial bees. Its definition (not shown) is simple, as it is only necessary to create `nb` bees randomly distributed inside the search space defined by the boundings `bo`. Once bees are initialized, we use an independent function `bees'` to deal with all the iterations of the ABC algorithm. As in the case of the main function `beesSEQ`, the auxiliary function `bees'` will also need a way to introduce randomness. This issue is solved by using function `split` to create new random generators. Finally, function `bees'` will also have, as an additional input parameter, the information about the best position found so far by the algorithm. Initially, it will be the best position found by the `initBees`, and after each step of the algorithm it will be updated with the information about the new best position found so far. The output of function `bees'` will be a tuple where the first element is the information about the best position found by the algorithm, and the second element is a list containing all the information about the final state of each bee.

```
beesSEQ rg sc nb it f bo = fst (bees' rg2 sc it f bo best1 initBees)
  where (rg1,rg2) = split rg
        initBees  = initializeBees rg1 nb bo f sc
        best1     = obtainBestBee initBees
-- Bee: Best local value, current position, scouting counter
type Bee = (Double,Position,Scouting)
```

Note that each bee contains three values. The first one is the best fitness found so far by the bee; the second one is its current position; and the third one is its scouting counter. Initially, it will be set to the value defined by the global scouting parameter. Then, it will be decremented as described in Section 3, so that when its value equals zero it will get abandoned and a new random position will be used for it.

Function `bees'` has a simple definition. It runs recursively on the number of iterations. If there are zero iterations to be performed then we just return as result a tuple containing the best result found so far and the list of input bees. Otherwise, we apply one iteration step (by using an auxiliary function `oneStepBee`) and then we recursively call our generic scheme with one iteration less. Note that the recursive call uses as input parameters the outputs obtained by the first iteration, that is, the best position found so far, `newBest`, and the new information about the current list of bees, `newBees`:

```
bees' _ _ 0 _ _ best bs = (best,bs)
bees' rg sc it f bo best bs = bees' rg2 sc (it-1) f bo newBest newBees
  where (rg1,rg2)          = split rg
        (newBest,newBees) = oneStepBee rg1 sc f bo best bs
```

Note that, for the sake of simplicity, we are assuming that the number of iterations is the only ending condition. In case we were interested in using other conditions like error distance we would only need to change the second line of the previous code adding a simple `if`: In case `newBest` is good enough we would return `(newBest,newBees)` and finishes; otherwise we would go on with the next call to `bees'`.

Let us now concentrate on how to perform each step. First, employed bees are updated by using function `moveEmployed`. Then, the resulting list of bees `employedBs` is used as input parameter of function `normalize`, that computes the normalized values `normFitness` of the fitness function for each bee, as described in Section 3.

Then, function `moveOnlookers` uses both values `employedBs` and `normFitness` to perform the work of the onlookers bees. The output of this function contains a new list of bees `onlookersBs`, which is used by function `abandoned` to perform the final phase of the algorithm. That is, function `abandoned` creates a new random bee for each bee whose scouting counter is over. The output of this function is a tuple where the first element is the information about the best position found so far, and the second parameter is a list containing the current information of all the bees. The next code shows the structure of the algorithm, where `rg1`, `rg2`, `rg3` are the random generators used by each phase of the algorithm:

```
oneStepBee rg sc f bo best bs = abandoned rg3 sc f bo onlookersBs
  where (rg1,rgAux) = split rg
        (rg2,rg3)  = split rgAux
        employedBs = moveEmployed rg1 sc f bo bs
        normFitness = normalize (map value employedBs)
        onlookersBs = moveOnlookers rg2 sc f bo normFitness employedBs
```

The definition of functions `moveEmployed`, `normalize`, `moveOnlookers`, and `abandoned` is trivial, but not shown due to lack of space. Anyway, the complete source code is publicly available at the URL http://antares.sip.ucm.es/prabanal/english/heuristics_library.html.

Once implemented the higher-order function dealing with the ABC algorithm, the programmer only needs to implement the fitness function of his current problem. In fact, it is not necessary to understand the internals of the higher-order Haskell definition. Moreover, the programmer can define the function, for instance, in C.

Regarding the efficiency of the implementation, we have performed some experiments with standard benchmark functions, in particular the sphere model, Schwefel's problem 2.22, generalized Schwefel's problem 2.26, generalized Rastrigin's function, and Ackley's function (see e.g. [18] for details about these functions). Each of the problems was executed ten times, and they were always solved in less than 5 seconds by using an Intel Core i3 3.20GHz processor, obtaining always a solution whose distance to the optimum was smaller than 10^{-7} .

5. Parallel Skeleton in Eden

In order to parallelize a program, the first task to be done is to identify time-consuming tasks that can be executed independently. In our case, the update of each employed bee could be done independently. Analogously, the update of onlookers bees could also be parallelized. Although the time needed to execute these tasks will usually be small, it will be executed many times and, in practice, most of the execution time of the program will be devoted to these tasks. In the case of employed bees, we have to apply the same operations to a list of bees. Thus, we could parallelize it by using the skeleton `map_par`. Analogously, we could also do the same in the case of onlookers bees. Thus, the only pieces of code to be modified would be functions `moveEmployed` and `moveOnlookers`. By doing so, we can create a simple skeleton to parallelize ABC algorithms. However, in many situations, this implementation will obtain poor speedups. In particular, in case the processors of the underlying architecture do not use shared memory, lots of communications will be needed among processes, dramatically reducing the speedup. The solution to this problem implies increasing the granularity of the tasks to be performed by each process.

As a first step, it would be more efficient to create only as many processes as processors available, and to fairly distribute the population among them. This can be done by substituting `map_par` by a call to `map_farm`, a parallel version of `map` that implements the idea of distributing a large list of tasks among a reduced number of processes.

By using `map_farm` the speedup will be improved. However, for each iteration of the algorithm `map_farm` would create a new list of processes, and it would have to receive and return the corresponding lists of bees.

A better solution to increase the granularity and to reduce the communications is inspired by Bulk Synchronous Parallelism [19]. We start splitting the list of bees into as many groups as processors available. Then, each group evolves in parallel independently during a given number of iterations. After that, processes communicate among them to redistribute the bees among processes, and then they go on running again in parallel. This mechanism is repeated as many times as desired until a given number of global iterations is reached.

In order to implement in Eden a generic skeleton dealing with this idea, we need to pass new parameters to function `beesPAR`. In particular, we need a new parameter `pit` to indicate how many iterations have to be

performed independently in parallel before communicating with the rest of processes. Besides, the parameter `it` will now indicate the number of parallel iterations to be executed, that is, the total number of iterations will be `it * pit`. Finally, we can add an extra parameter `nPE` to indicate the number of processes that we want to create (typically, it will be equal to the number of available processors). Thus, the new type of `beesPAR` will be:

```
beesPAR :: RandomGen a => a          -- Random generator
      -> Scouting                    -- Scouting counter parameter
      -> Int                         -- Number of bees to be used
      -> Int                         -- Iterations in each parallel step
      -> Int                         -- Number of parallel iterations
      -> Int                         -- Number of parallel processes
      -> (Position -> Double)        -- Fitness function
      -> Boundings                  -- Search space boundaries
      -> (Double, Position)          -- Value and position of best fitness
```

Before dealing with the main function `beesPAR`, we define the function describing the behavior of each of the processes of the system. In addition to receive a parameter `rg` for creating random values, each process needs to receive a parameter with the scouting counter `sc`, the number of iterations to be performed in each parallel step `pit`, the fitness function `f`, and the boundings of the search space `bo`. Besides, it also receives through input channels two parameters: the best position found so far, and a list of list of bees. For each inner list of bees, the process will apply `pit` iterations of the algorithm, returning their state after such iterations. Recall that in Eden list elements are transmitted through channels in a stream-like fashion. This implies that each process will receive a new list of bees through its second input channel right before starting to compute a new parallel step. The output of the process is a tuple where the first parameter is the best position found so far, and the second parameter is a list containing the list of bees computed after each parallel step of the process. The implementation is the following:

```
beesP rg sc pit f bo (best,[]) = (best,[])
beesP rg sc pit f bo (best,(bs:bss)) = (finalBest,newBs:newBss)
  where (rg1,rg2)          = split rg
        (newBest,newBs)    = bees' rg1 sc pit f bo best bs
        (finalBest,newBss) = beesP rg2 sc pit f bo (newBest,bss)
```

Note that the definition of the process is simple. It is defined recursively on the structure of the second input channel. That is, for each list `bs` of bees received, the process uses exactly the same `bees'` function used in the sequential case, returning the best position found so far (`newBest`) and the state of the bees (`newBs`) after `pit` iterations. Then, `newBs` is returned through the second output channel, so that the main function `beesPAR` can shuffle them with the bees computed by the rest of processes, so that new input lists of bees are computed for the next global step. When the main function finishes sending the list of lists of bees through the input channel (represented by an empty list `[]`), the process also finishes its execution, and it sends through its first output channel the best position found by the process.

Finally, we define the main function `beesPAR`. As in the sequential version `beesSEQ`, now we also need to create a list of random generators `rgs`, one for each process. The first step it must perform is also the same as in the sequential case, that is, bees are initialized by using the same sequential function `initializeBees`. The difference is that now we instantiate `nPE` copies of process `beesP`. Each of the copies receives the main input parameters of the algorithm (scouting counter, fitness function, etc.), and it also receives its own list of tasks (`pins!!i`). Each element of the list of tasks contains an input list of bees, that will be processed by `beesP` during `pit` iterations (see Figure 1). The output of each process `beesP` will be a tuple containing the best position found and a list containing the list of bees computed after each parallel step. Note that these output lists of bees computed after each parallel step must be used as input for the next parallel step. This is done by function `redistribute`. Finally, note also that the global output of function `bestPAR` only needs to consider the list of best solutions found by each process, and then selects the maximum of them. The source code is the following:

```
beesPAR rg sc nb pit it nPE f bo = maximum bests
  where initBees = initializeBees rg nb bo f sc
```

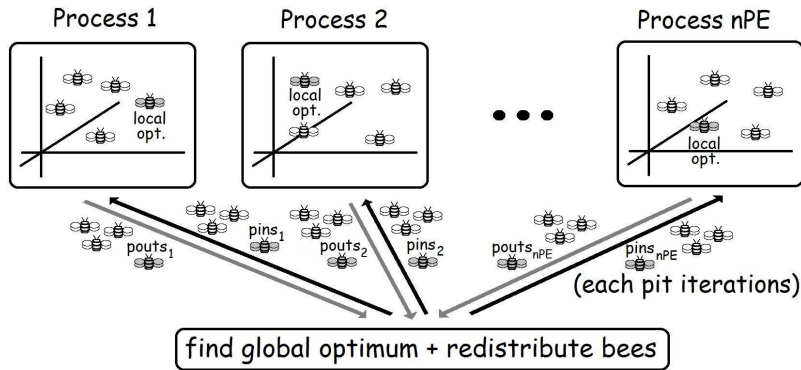


Fig. 1. Basic structure of the skeleton.

```

initBest = obtainBestBee initBees
rgs      = tail (generateRGs (nPE+1) rg)
pouts    = [process (beesP (rgs!!i) sc pit f bo) # (initBest, take it (pins!!i))
            | i <- [0..nPE-1]] 'using' spine
(bests, beesss) = unzip pouts
pins         = redistribute nPE initBees beesss

```

Let us remark that, in order to convert a sequential ABC algorithm into the corresponding parallel program, the programmer only has to change a call to function `beesSEQ` by a call to function `beesPAR`, indicating appropriate values for parameters `pit`, `it` and `nPE`. In fact, the only programmer effort is due to selecting a reasonable value for `pit`. Small `pit` values would reduce the granularity of the tasks, degrading the speedup, while very large `pit` values would reduce the possibility to communicate good solutions among processes. In fact, in the limit we could use `it = 1` and `pit` equals to the total number of iterations to be performed. In that case, we would have `nPE` totally independent groups of bees searching for a solution.

Note that the programmer does not need to deal with the details of the parallelization. In fact, it is not even necessary that the programmer understands the details of the parallel implementation, provided that he understands the type interface of function `psPAR`. Actually, the only function to be developed by the programmer is the fitness function. Moreover, it is important to recall that Eden programs can interact with other programming languages. In particular, C code can be encapsulated inside a Haskell function. Hence, Eden can be used as a coordination language dealing with the parallel structure of the program, whereas the core of the fitness function could be implemented in a computation language like C.

Let us finally comment that it is possible to provide more versions of the ABC skeleton, so that the programmer can select the one that better fits his necessities. For instance, if the parallel environment to be used is not homogeneous, then the distribution of tasks among processors should take into account the characteristics of each processor. Let us consider a simple environment with two processors where one of them is much faster than the other. In that case, it would not be a good idea to assign half of the bees to each processor, because the faster one would finish its assignment in less time than the other one. Thus, it would be idle during the rest of the time, waiting until its counterpart finishes its tasks and thus reducing the overall performance of the system. Obviously, the problem is the same when the number of processors is larger and not all of them have the same speed.

The solution is simple: the number of bees to be assigned to each processor should depend on the relative speed of all of them. Thus, instead of an integer denoting the number of available processors, the new skeleton has a new parameter containing a list of numbers denoting the speed of each processor. Note that, given the list, we also know the number of processors, so we can forget the `nPE` parameter. The type of the new skeleton is the following:

```

beesPARh :: RandomGen a => a          -- Random generator

```



```

-> Scouting                -- Scouting counter parameter
-> Int                     -- Number of bees to be used
-> Int                     -- Iterations in each parallel step
-> Int                     -- Number of parallel iterations
-> [Double]                -- Speed of processors
-> (Position -> Double)    -- Fitness function
-> Boundings               -- Search space boundaries
-> (Double,Position)       -- Value and position of best fitness

```

Regarding the implementation, we only need to modify the definition of `beesPAR` to include a new way to create the list of lists of bees pins by unshuffling the initial bees among processes, in such a way that their speeds are taken into account:

```

beesPARh rg sc nb pit it speeds f bo = maximum bests
  where nPE = length speeds
        initBees = initializeBees rg nb bo f sc
        initBest = obtainBestBee initBees
        rgs      = tail (generateRGs (nPE+1) rg)
        pouts    = [process (beesP (rgs!!i) sc pit f bo) # (initBest,take it (pins!!i))
                     | i<-[0..nPE-1]] 'using' spine
        (bests,beesss) = unzip pouts
        pins          = redistributeRelative speeds initBees beesss

```

The new function `redistributeRelative` is based on an auxiliary function `shuffleRelative` that first computes the percentage of tasks to be assigned to each process, and then it distributes the tasks by using function `splitWith`:

```

shuffleRelative speeds tasks = splitWith percentages tasks
  where percentages = map (round.(m*).(/total)) speeds
        total      = sum speeds
        m          = fromIntegral (length tasks)
splitWith [n] xs = [xs]
splitWith (n:ns) xs = firsts:splitWith ns rest
  where (firsts,rest) = splitAt n xs

```

Note that we do not need to change any other definition of the previous skeleton. In particular, the definition of process `beesP` is exactly the same.

Next we present the speedups obtained by using our skeleton. In contrast to other approaches where a shared memory environment is assumed (see e.g. [20]), in order to analyze the parallel performance we use two computers which are connected through the intranet of our university and are physically located in different buildings. Both computers use the same Linux distribution (Debian), the same MPI library (OpenMPI 1.4.2), and have analogous Intel Core i3 Duo processors. Thus, we perform parallel experiments with up to four cores. Note that when using only 2 processors both of them are inside the same computer, but in the case of using 3 or 4 processors we are actually using both computers. The speedups are computed by comparing the execution time with the time of the sequential version. For each number of processors, ten executions were performed, and the average time was used. The speedups obtained when solving Schwefel's problem 1.2 with 75000 iterations of the algorithm and using `pit = 1000` are 1.9 (with 2 processors), 2.7 (with 3 processors), and 3.3 (with 4 processors). Let us remark that, once the skeleton is defined, the programming effort needed to parallelize a concrete problem is negligible, while the obtained speedups are acceptable.

6. Conclusions and Future Work

In this paper we have shown how to provide a generic parallelization of ABC by using the parallel functional language Eden. We have presented a concrete parallel skeleton that can be instantiated with any concrete fitness

function. Thus, the programmer only has to provide such function and the number of iterations to be performed, whereas all the low-level details of the parallelization are done automatically by the skeleton and the underlying runtime system. Moreover, different parallel implementations of ABC can be provided to the user, so that he can select the one that better fits his necessities.

Experiments have also been presented, showing that (i) the sequential version of our algorithm obtains good solutions in reasonable time; (ii) the parallel skeleton provides reasonable speedups; and (iii) the programming effort is negligible once the skeleton has been provided. Note that we do not claim that we reach the optimal speedup, but we obtain a *reasonable* speedup at a *low* programming effort.

Our aim in the near future is to provide a library of Eden skeletons for the most common bioinspired metaheuristics. In fact, in previous work [21, 22], Eden skeletons were implemented to parallelize both genetic algorithms and Particle Swarm Optimization. By developing a large library of parallel versions of common metaheuristics, we will simplify the task of using them in functional settings, and we will also simplify the task of improving their performance by means of parallelizing them. Moreover, as several metaheuristics will be provided in the same environment, the programmer will be able to check more easily what metaheuristics fits better for each concrete problem.

In addition to extend our library to deal with other evolutionary computation methods (like Ant Colony Optimization or River Formation Dynamics), as future work we are particularly interested in studying hybrid systems combining different metaheuristics.

References

- [1] D. Karaboga, An idea based on Honey Bee Swarm for Numerical Optimization, Tech. Rep. TR06 (2005).
- [2] D. Karaboga, B. Basturk, On the performance of artificial bee colony (ABC) algorithm, *Applied Soft Computing* 8 (1) (2008) 687 – 697.
- [3] D. Karaboga, B. Akay, A survey: algorithms simulating bee swarm intelligence, *Artificial Intelligence Review* 31 (1-4) (2009) 61–85.
- [4] M. Cole, Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming, *Parallel Computing* 30 (2004) 389–406.
- [5] P. W. Trinder, K. Hammond, J. S. M. Jr., A. S. Partridge, S. L. P. Jones, GUM: a portable parallel implementation of Haskell, in: *Programming Language Design and Implementation, PLDI'96*, ACM Press, 1996, pp. 79–88.
- [6] U. Klusik, R. Loogen, S. Priebe, F. Rubio, Implementation skeletons in Eden: Low-effort parallel programming, in: *Implementation of Functional Languages, IFL'00, LNCS 2011*, Springer, 2001, pp. 71–88.
- [7] N. Scaife, H. S., G. Michaelson, P. Bristow, A parallel SML compiler based on algorithmic skeletons, *Journal of Functional Programming* 15 (4) (2005) 615–650.
- [8] M. Hidalgo-Herrero, Y. Ortega-Mallén, F. Rubio, Analyzing the influence of mixed evaluation on the performance of Eden skeletons, *Parallel Computing* 32 (7-8) (2006) 523–538.
- [9] S. Marlow, S. L. Peyton Jones, S. Singh, Runtime support for multicore Haskell, in: *Int. Conference on Functional Programming, ICFP'09*, ACM Press, 2009, pp. 65–78.
- [10] G. Keller, M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, B. Lippmeier, Regular, shape-polymorphic, parallel arrays in Haskell, in: *Int. Conference on Functional Programming, ICFP'10*, ACM, 2010, pp. 261–272.
- [11] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, F. Rubio, Parallelism abstractions in Eden, in: *Patterns and Skeletons for Parallel and Distributed Computing*, Springer, 2002, pp. 95–128.
- [12] S. L. Peyton Jones (Ed.), *Haskell 98 language and libraries : the revised report*, Cambridge University Press, 2003.
URL <http://www.worldcat.org/isbn/9780521826143>
- [13] P. W. Trinder, K. Hammond, H.-W. Loidl, S. L. Peyton Jones, Algorithm + Strategy = Parallelism, *Journal of Functional Programming* 8 (1) (1998) 23–60.
URL <http://www.dcs.glasgow.ac.uk/~hwloidl/publications/strategies.ps.gz>
- [14] G. Beni, J. Wang, Swarm intelligence in cellular robotic systems, in: *NATO Advanced Workshop on Robotics and Biological Systems*, 1989.
- [15] E. Bonabeau, M. Dorigo, G. Theraulaz, *Swarm intelligence: from natural to artificial systems*, Oxford University Press, Inc., 1999.
- [16] J. Kennedy, R. Eberhart, *Swarm intelligence*, Morgan Kaufmann Publishers Inc., 2001.
- [17] J. Kennedy, *Swarm intelligence*, in: *Handbook of Nature-Inspired and Innovative Computing*, Springer US, 2006, pp. 187–219.
- [18] X. Yao, Y. Liu, G. Lin, Evolutionary programming made faster, *IEEE Trans. Evolutionary Computation* 3 (2) (1999) 82–102.
- [19] D. Skillicorn, J. Hill, W. McColl, Questions and answers about BSP, *Scientific Programming* 6 (3) (1997) 249–274.
- [20] H. Narasimhan, Parallel artificial bee colony (PABC) algorithm, in: *World Congress on Nature Biologically Inspired Computing, NaBIC'09, IEEE*, 2009, pp. 306–311.
- [21] A. de la Encina, M. Hidalgo-Herrero, P. Rabanal, F. Rubio, A parallel skeleton for genetic algorithms, in: *IWANN'11, LNCS 6692*, Springer, 2011, pp. 388–395.
- [22] P. Rabanal, I. Rodríguez, F. Rubio, A functional approach to parallelize particle swarm optimization, in: *Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB'12*, 2012.